Portable, Parallel Transformation: Distributed-Memory Approach

LAWRENCE A. COVICK* and KENNETH M. SANDO†
Department of Chemistry, University of Iowa, Iowa City, Iowa 52242

Received 28 June 1994; accepted 8 August 1995

ABSTRACT

The four-index transformation has a high ratio of data transfer to computation making it a potential "bottleneck" for parallel correlation energy determination. We present formulas for the communication times on different parallel architectures for an algorithm that is primarily designed for distributed-memory machines. We also implemented the algorithm on two shared-memory parallel computers, the Encore Multimax and the Alliant FX-8, and measured wall clock times for several problem sizes and processor configurations. © 1996 by John Wiley & Sons, Inc.

Introduction

The four-index transformation is one of the more computationally intensive parts of electronic structure programs. It is also a small piece of code that is comparatively simple. For these reasons it is an obvious choice for parallel algorithm development. At the same time it is a challenge to the programmer and a possible "bottleneck" because it contains a high ratio of data transfer to computation.

Our approach is twofold. First, we are looking for truly portable programs that will run on a variety of parallel computers. Second, we want to investigate a distributed-memory approach to programming where the problems are somewhat different than with the more common shared-memory approach. (Instead of memory contention, the major problem is interprocess communication,[1] which depends upon the communication topology of the computer.) The portable FORTRAN message passing macros[2] developed at Argonne National Laboratory are ideal for this effort. These macros can simulate distributed systems with various topologies while being employed on a shared-memory machine. To do this a region of shared memory is divided into P equal partitions, with one partition assigned to each simulated processor. The messages go from process to process and follow the chosen topology so that multihop communication is simulated. No artificial latencies in communication time were added.

In addition, as we have previously stated,[3] some of the ideas in distributed algorithms might work well in a shared-memory environment. This work

* Present address: Department of Chemistry, Athens State College, Athens, AL.
† Author to whom all correspondence should be addressed.

CCC 0192-8651 / 96 / 080992-10

is a summary of our results with some shared-memory systems.

Methods

The development of our trial algorithm was done primarily on two machines: an Alliant FX-8 located at Argonne and an 18 processor Encore Multimax located at the University of Iowa. The results shown are also from these two machines. The algorithm presented here is not an optimal four-index transformation algorithm in general, is not for a particular parallel computer architecture or for any individual machine, but it is adaptable to many architectures and is useful in testing new ideas.

This algorithm is different than the one published in our last article[3] and from other distributed memory algorithms.[4] It also differs from shared-memory algorithms. In shared-memory algorithms the problem is treated as the transformation of a series of two-dimensional arrays. The transformation takes essentially two steps with a sort and usually disk storage of integrals in between.[5-8] In the distributed-memory algorithms of Whiteside et al.[4] and our previous one,[3] the transformation was done as a sequence of a two-dimensional transformation followed by two one-dimensional transformations. In addition, each of these was performed within a process or task (algorithmic constructs) that contained one or more two-dimensional arrays. These arrays had to exchange data with each other after the two-dimensional transformation and after the first one-dimensional one. (For further details see ref. 3.)

The algorithm presented here treats the data as being in N three-dimensional arrays (where N is the number of basis functions) housed within a certain number of processes, P, where at the current stage of development N is an integer multiple of P (i.e., a "load-balanced" system). The transformation can then be seen as a series of three-dimensional transformations that are done concurrently by the processes (using only their local memory), followed by interprocess communication, and then a one-dimensional transformation at each process. This means that instead of two interprocess communication steps there is only one, corresponding to a rearrangement of integrals based on one of the four indices.

The basic philosophy of the message passing remains the same as in our previous algorithm,[3] but the actual message passing differs because of the length of the messages that can be sent with this algorithm. Since it is designed for N processors, whole matrices can be passed (these are of course triangular if one utilizes the permutational symmetry of the integrals) instead of vectors. This means that the messages should be long enough so that any start-up time is negligible and our process of "pipelining" the messages in the previous work is unnecessary. The message passing is handled similarly, except that messages only have to go in one direction rather than two.

There are two other differences between this algorithm and our previous one. Both algorithms use one of the permutational symmetries of the integrals, but this algorithm uses it throughout the entire transformation while the previous one used it only for the transformation of the second index. If t_{comp} is the computation time in units of floating point multiply operations (FPMOs) it is now equal to $5N^5/2$ versus the $7N^5/2$ of our previous algorithm. This algorithm also does not have the same final distribution of integrals as our previous one, so the "super-integrals" cannot be created locally. This is not a major concern since a communication step similar to those of our previous algorithm can be used to create that distribution in a small amount of time, especially on high bandwidth architectures.[3]

Results

FORMULAS

We first present formulas to predict communication times, parallel efficiencies, and under what conditions to expect the speedup to be linear in the number of processors.

The formulas in our previous work[3] were the result of basing the algorithm on the concept of pipelining. This is essentially a bundling of messages for various processors with each processor taking its message from the bundle as it passes. This concept makes less sense as the number of processors is decreased because the messages would become longer (necessitating larger buffers on each processor) and the start-up times for the messages would become a much smaller part of the total communication time. In this work we concentrate on the case where $P \simeq N$. This means that the actual code will not be pipelined. In addition, our earlier pipelined algorithm was not necessarily optimal for all of the architectures investi-

gated but was used as a demonstration of an algorithm that could take advantage of a particular architecture.

We would now like to present a method for calculating optimal wall clock communication timings, in units of words of data transferred, for the mapping of our algorithm onto homogenous host graphs (i.e., actual processor topologies where every node is connected to the same number of neighbors). As in our previous work these "timings" neglect all machine dependent parameters and are represented by $t_{I/O}$. Since they are machine independent and in units of words of data transferred, they must be divided by the data transfer rate of the machine in question to get the wall clock time in seconds for that machine. For example, to convert $t_{I/O}$ to seconds for the Encore Multimax, one would divide by 5.6 MB/s (see below).

The method that we use to calculate the optimal communication times can be summarized as, "pipelining all of the data but only allowing the data to be sent along one dimension of the host graph at a time."[3]

Optimal communications times can be obtained with the following steps:

1. Assume that all processors can send data simultaneously, that each processor can send and receive data simultaneously, and that there is two-way communication between processors.
2. The entire host graph is divided into groups of processors with each group directly connected to the sending processor.
3. The groups are assigned such that there is no shorter path to any member that goes through nonmembers of the receiving group.
4. Each group contains the maximum number of processors that satisfy condition 3 (see Fig. 1a, c).
5. Each sending processor sends all available data designated for those groups connected to it in the dimension containing the largest receiving group.
6. After the receiving group receives its message, the receiving member becomes a sending processor for the next message. It then forms new groups for the next message using steps 2, 3, and 4 but only for processors for which it now possesses data (see Fig. 1b, d).
7. Steps 5 and 6 are continued until all of the data arrives at its final destination.

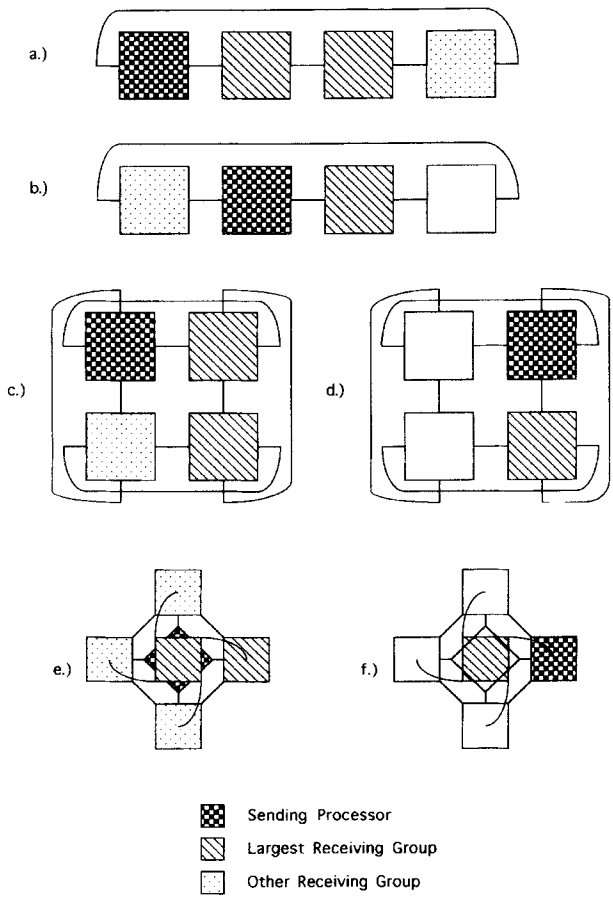

FIGURE 1. Method of finding the optimal formula for a homogenous host graph. Processors are indicated by squares and groups of processors are designated by squares with the same pattern. Three topologies are shown, (a) and (b) a ring, (c) and (d) a square array, and (e) and (f) an octahedron. The sending processor sends messages to the receiving groups in the dimension of the largest receiving group [(a), (c), and (e)]. The processor in the receiving group that receives the message then becomes a sending processor for all of the data that it now possesses [(b), (d), and (f)].

8. The total time is just the sum of the times for the individual messages divided by the number of dimensions that can communicate simultaneously.

In Table I optimal $t_{I/O}$ formulas for our algorithm on a number of host graphs are shown. Unlike the formulas derived in our previous work, the formulas for $t_{I/O}$ are for total time (instead of one communication step) and the mapping is onto N tasks instead of N^2. These are in turn mapped onto P processes, where $P = N/n$ (n is an integer). Since N is restricted to being an integer

TABLE I.
Formulas for Optimal Communication Times for Four-Index Transformation Algorithm on Several Parallel Computer Architectures.

Architecture	$t_{I/O}$ (Words of Data)[a]
Ring	$t_{I/O} = \frac{M}{2P^2}X(X+1)$
Square array	$t_{I/O} = \frac{M}{2P^{3/2}}Y(Y+1)$
Sequential communicating hypercube	$t_{I/O} = \frac{M}{2P}\log_2(P)$
Parallel communication hypercube	$t_{I/O} = \frac{M}{2P}$
Complete graph	$t_{I/O} = \frac{M}{P^2}$

[a] N is the number of basis functions, P is the number of processors, $P = \mathcal{O}(N)$ and two-way communication is assumed between processors. $M = N^3(N+1)/2$; X = floor $(P/2)$; and Y = floor $(P^{1/2}/2)$.

multiple of P, all of the formulas are for load-balanced cases. The architectures investigated include two types of hypercubes, one that can communicate along all dimensions simultaneously (PCH) and one that can only communicate one dimension at a time (SCH), and a square array as defined by Whiteside et al.[4] For further explanation of each of the architectures see ref. 3.

It can be seen from Table I that different topologies yield different communication times. This is because different amounts of data can be sent simultaneously, depending on the number of dimensions of the host graph. In addition, the distance traveled by the data is architecture dependent; therefore, it will take different amounts of time for all of the data to reach its final destination, depending on the architecture employed. For example, a ring will have a $t_{I/O}$ $\mathcal{O}(P^2)$ greater than a complete graph because the communication bandwidth is $\mathcal{O}(P)$ smaller and an average message must travel $\mathcal{O}(P)$ further to get to its destination. Another point is that the formulas as written do not give $t_{I/O} = 0$ when $P = 1$ for a complete graph or a PCH. This is because all of the formulas have a factor of $(1 - 1/P)/(P - 1)$ that has been reduced to $1/P$ for simplicity. Each of these factors and therefore each formula is undefined when $P = 1$.

As stated above, if an actual algorithm bundled messages for interprocessor communication, the amount of memory required on each processor would be excessive for messages of the length required when working with $\mathcal{O}(N)$ processors. While the method for finding the optimal formulas uses this approach, an algorithm can be devised for each architecture where every node sends only $\mathcal{O}(N^2)$ integrals at a time, uses no more than $\mathcal{O}(2dN^2)$ (d is the number of dimensions of the host graph) words of memory for communication buffers on each node, and has the same communication time.

For example, with a ring topology (such as that implemented below, see Results) to pass a message to the furthest processor [myid + floor$(P/2)$] would require the following steps.

```
newhom = myid + floor(P/2)
Do i = 1, N(N + 1)/2
  Buffer 1(i) = Ints(newhom, i)
enddo
Do i = 1, floor(P/2)
  if(i odd) then
    SEND buff1 to myid + 1
    RECEIVE buff2 from myid − 1
  else
    SEND buff2 to myid + 1
    RECEIVE buff1 from myid − 1
  endif
enddo
```

Each block of integrals is sent to its destination in a similar fashion. If instead the method used to find the formulas was literally implemented, then all of the data would be sent at once and each processor would have to sort the data received and send each portion in the proper direction so that all of the data was taking the shortest possible path to its destination.

Because the present algorithm uses more permutational symmetry than our previous one, there are fewer operations and less data transferred. This algorithm will be faster in load-balanced cases and in most nonload balanced cases in spite of the fact that it maps the integrals onto N tasks while our previous one used N^2 tasks. However, with many processors and few basis functions, specifically, $P \geq 0.4N$ (we are assuming that each processor contains only one process so P now is also the number of processors used), the N^2 algorithm will be faster for low processor computational speeds and/or high interprocessor data transfer rates. In the range $0.4N \leq P < N$, cross-over points were calculated for realistic values of P and N with the values of the ratio of the processor com-

putational speed (in Mflops) to the interprocessor data transfer rate (in MB/s) ranging between 0.5 and 4.

The least favorable comparison between the present N algorithm and the N^2 algorithm occurs for non-load-balanced cases with $P \simeq N$. Distributing the integrals in N blocks can give some processors as many as twice the calculations of other processors. The effect is most pronounced when $P = N - 1$. In this instance t_{comp} for one processor is twice that for all of the others. This difference overwhelms the slight communication disadvantage that an N^2 distribution would have due to an additional communication step between the second and third index transformations required since each processor no longer has all the integrals necessary for the transformation of the third index. [Using the method explained below to obtain a formula for a ring topology (see Results), the additional communication time is given by $t_{I/O} = \text{floor}(p/2)N(N+1)/2$.] At the other extreme, if $P = N + 1$, the wall clock computational time will be the same for both algorithms, but the N^2 one will now have a slightly longer total wall clock time because again it would have the additional communication step and time given above.

Trial Systems

The algorithm that was implemented on the two shared-memory computers was limited to communication in one direction at a time, so the timing estimate is given by the following formula

$$t_{I/O} = \frac{M}{P^2}(X + Q), \tag{1}$$

where $X = \text{floor}(P/2)$,

$$M = \frac{N^3(N+1)}{2}, \tag{2}$$

and

$$Q = \text{ceil}\left(\frac{\text{ceil}(P/2)}{X}\right) - 1. \tag{3}$$

Although the implemented algorithm was not pipelined as defined above, its time is equivalent to that for a pipelined algorithm restricted to one-way communication because each processor sends its messages one at a time to receivers that are at similar positions relative to the sender.

The efficiency of a parallel computer can be defined as the fraction t_{scomp}/t_{stotal} where t_{scomp} is the computation time and t_{stotal} is the total time, and both are measured in seconds. We can define the load-balanced parallel efficiency (LPE) for a system that does not overlap communication with computation by

$$\text{LPE} = \frac{t_{comp}/\text{Mflops}}{t_{comp}/\text{Mflops} + 8t_{I/O}/\text{TR}} = \frac{t_{comp}}{t_{comp} + 8\left(\frac{\text{Mflops}}{\text{TR}}\right)t_{I/O}}, \tag{4}$$

where t_{comp} and $t_{I/O}$ are the same as above, Mflops is the processor speed in millions of floating point operations per second, and TR is the data transfer rate between processors (interprocessor bandwidth) in units of megabytes per second.

Since we are only interested in estimates, we adopt the manufacturer's Mflops rating for the processors. Because the algorithm uses an equal number of multiplications and additions, and since t_{comp} is in units of FPMOs, we will scale it by a factor of 2.5 for scalar machines (where we have used the conversion factor 1 multiplication equals 1.5 additions) and by 2 for vector processors (assuming that additions and multiplications take the same amount of time). (For the Encore we determined experimentally that the ratio should be 2.25 instead of 2.5, but it turns out that this has no effect on the results from our formulas found below.) This is different from our previous work where we used the definitions used by Cisneros et al.[8] for Mflops and TR.

On shared-memory machines we estimate TR as the memory bus bandwidth divided by the number of processors, since the memory has been divided up among processes that at this stage of development correspond to the number of processors. We also assume that communication is not overlapped with computation. This is a worst case scenario and it is sufficient to see if this will produce near linear speedups. (We dealt with the overlap of communication with computation in our previous work.[3])

For a given architecture, if either N or P is held constant, a graph of the inverse of the LPE (ILPE) versus Mflops/TR produces a family of straight lines. These all have the same intercept, which is the limiting value of the ILPE of 1. If both N and P are held constant, then the plot will consist of a family of lines showing the dependence of the ILPE on the Mflops/TR for various architectures. This is done in Figure 2 for $N = 256$ and $P = 256$.

The region under the dashed line corresponds to high efficiencies (LPE > 90%). Figure 2 thus gives an idea of how the efficiency is affected by architecture. All of the plots on Figure 2 are for the formulas from Table I, so they are optimal cases for our algorithm.

From Figure 2 it can be seen that the complete graph gives far higher efficiencies for a particular Mflop/TR than any of the other architectures. It can also be seen that for this problem and machine size the order of the other architectures is PCH > square > SCH > ring. The only surprise is the relative ordering of the square and the SCH. From the formulas of Table I the square has a dependence on $P^{-1/2}$, and the SCH has a dependence on $(\log_2(P))/P$. From this one can expect that for very large problems and machine sizes the SCH will have a higher LPE (lower ILPE), but because of the constants this does not happen until about 2048 processors (an "eleven cube") are used.

Even though the complete graph has the highest LPEs of any architecture, it would be difficult to actually build one with more than a few processors because of the high number of interconnections necessary. Figure 3 shows what one could expect from a complete graph used for a small system of processors ($P = 12$). One would also expect a shared memory system without memory conflict to have a similar result. It predicts that for a problem size of 156 basis functions the processor speed in Mflops could be 140 times the communication rate in MB/s and an LPE of 90% would be attained. This again is an optimal time that assumes simultaneous two-way communication between processes.

The slowest of the proposed architectures is the ring, and the trial ring algorithm that we actually implemented was not optimal. Figure 4 is a plot of a ring whose $t_{I/O}$ is governed by Eq. (2). In this case for $N = 156$ an LPE of 90% can be attained if the Mflops/TR ≤ 3; and if Mflops $\approx$ TR, a 12 processor machine is predicted to run with near-linear speedup for all but the smallest of problems.

Figure 5 shows what happens when this same message passing architecture is used for a constant problem size ($N = 128$) but the number of processors is allowed to vary. The figure shows that as the number of processors increase for a given problem size, the Mflop/TR must decrease to maintain near-linear speedup. By the time $P = 64$, the data transfer rate must be twice the processor speed to obtain near-linear speedup.

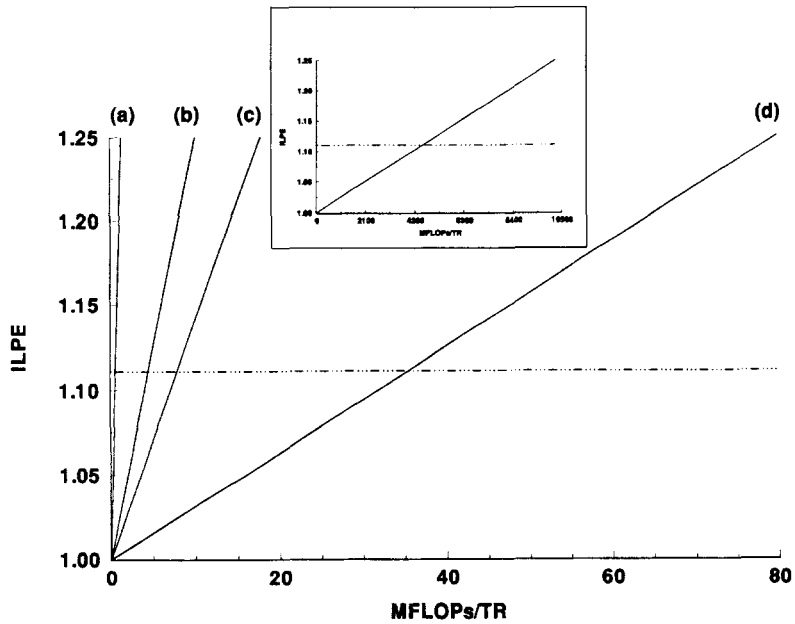

FIGURE 2. Plot of the inverse load-balanced parallel efficiency (ILPE) (with near-linear speedups obtained in the region under the dashed line) versus the ratio of processor speed to interprocessor data transfer rate (Mflops / TR, with TR in MB / s) for a constant problem and machine size (256 basis functions and processors) using the following host graph architectures: (a) a hypercube that can communicate in all dimensions simultaneously (parallel communicating hypercube or PCH); (b) a square array; (c) a hypercube that can only communicate in one dimension at a time (sequential communicating hypercube, or SCH); (d) a ring; and in the inset, a complete graph architecture.

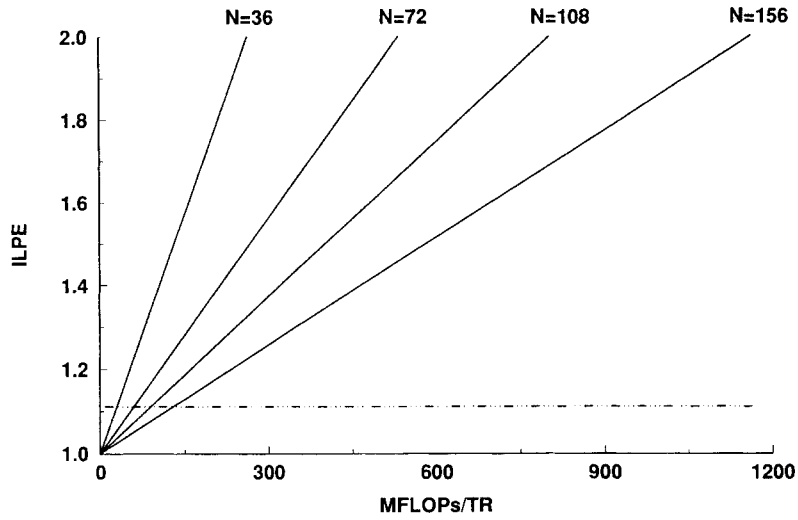

FIGURE 3. Plot of the inverse load-balanced parallel efficiency (ILPE) versus the ratio of the processor speed (in Mflops) to the interprocessor data transfer rate (TR, in MB / s) for various problem sizes (*N* is the number of basis functions) using a 12 processor complete graph as the host architecture with the region under the dashed line corresponding to near-linear speedups.

Since the Alliant and Encore computers on which we implemented our algorithm only had 8 and 18 processors, respectively, it should be appropriate to see what our worst case scenario formula gives for the LPEs of some trial problem sizes and later to compare those with experiment. What will our formulas predict for the LPE if we use a ring of processes as the algorithmic architecture (guest graph) implemented on a shared-memory machine's architecture (host graph), and if we use the manufacturers' numbers for the values of Mflops and TR? For the Encore we used Mflops = 0.25 and TR = (100 MB/s)/(18 processors) or 5.6 MB/s. For the Alliant we used Mflops = 11.8 and TR = (188 MB/s)/(8 processors), or 23.5 MB/s.

For the Encore Multimax with 18 processors and 54 basis functions, we predict a LPE of over 99%, and for the Alliant FX-8, with 8 processors

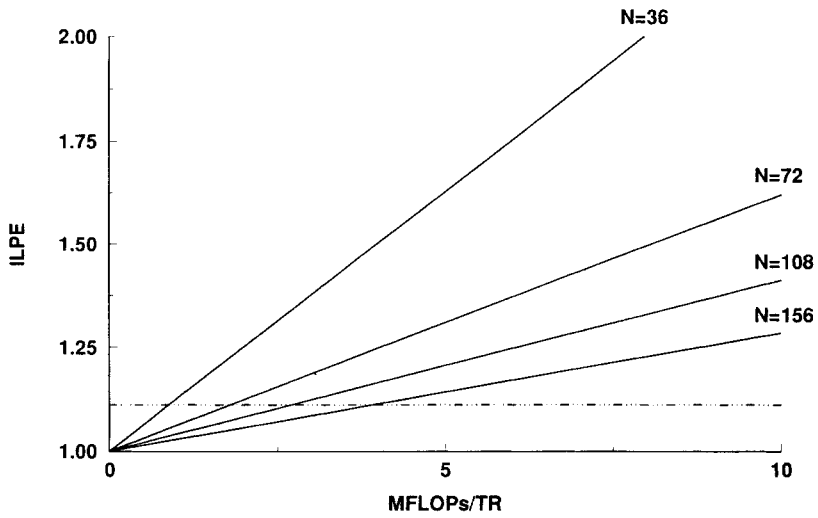

FIGURE 4. Plot of the inverse load-balanced parallel efficiency (ILPE) versus the ratio of the processor speed (in Mflops) to the interprocessor data transfer rate (TR, in MB / s) for various problem sizes (*N* is the number of basis functions) using a 12 processor ring as the host architecture with the region under the dashed line corresponding to near-linear speedups.

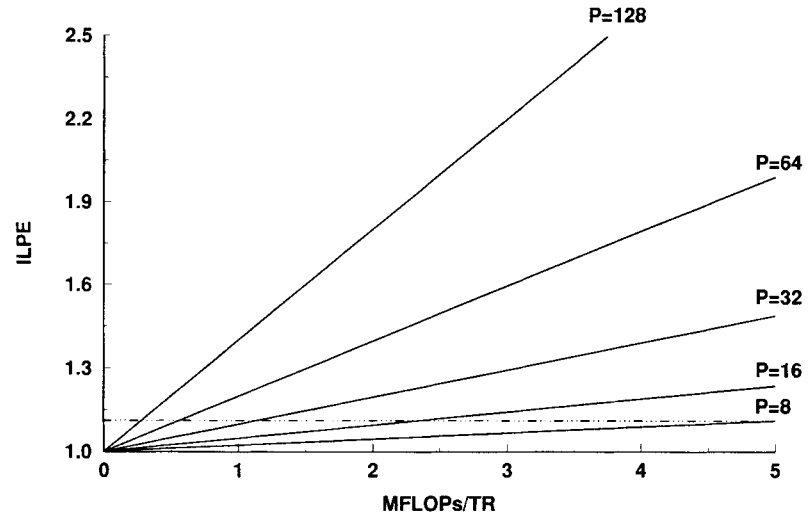

FIGURE 5. Plot of the inverse load-balanced parallel efficiency (ILPE) versus the ratio of the processor speed (in Mflops) to the interprocessor data transfer rate (TR, in MB / s) for a constant problem size (128 basis functions) using ring host graphs of various sizes (P is the number of processors in the ring) with the region under the dashed line corresponding to near-linear speedups.

and 48 basis functions, an LPE of 97%. If the problem size is increased to $N = 72$, then both give an LPE of over 99%. It should be noted that in both cases we used the rated peak Mflops for the processors, and for the Alliant we used a ratio of 1 for the relative speeds of an FPMO to an FPAO (floating point addition operation). For a given value of TR all of these should produce a worst case scenario for the LPE. Therefore, our estimates of the LPEs should be worst case scenarios based solely on the amount of computation and communication that a given processor has to do and the manufacturers values for the peak performance of these functions by a processor (assuming all processors are running simultaneously with equal priority). Since our formulas predict essentially linear speedup for both machines, for this case they will also predict linear speedups for the other higher bandwidth guest graphs (although the calculations may be more complicated).

EXPERIMENTAL

With the exception of the single process (processor) results, all results were obtained when our account was listed as the sole user of the machine; and the ones indicated were done when we were given sole access to the machine. All of the times listed are wall clock times that are given by the Unix "time" commands as elapsed time output.

Encore Multimax Results

For the Encore Multimax we looked at the following: the effect of varying P for a given value of N, the effect of varying N for a given P, and the effect of varying the guest graph for a given value of N and P. The results for the Encore are given in Table II.

TABLE II.
Wall Clock Times for Algorithm on Encore Multimax.

N	P	Architecture	t (s)	Speedup	PE (%)[a]
45	1	—	7472	—	—
45[a]	3	Ring	2615	2.86	95
45[a]	5	Ring	1561	4.79	96
45[a]	9	Ring	879	8.50	94
45[a]	15	Ring	553	13.51	90
54	1	—	24430	—	—
54[a]	18	Ring	1540	15.86	88
54	18	c Graph	1484	16.46	91
72	1	—	81881	—	—
72[a]	18	Ring	5596	14.63	81
72[a]	18	c Graph	5539	14.78	82
18[a]	18	Ring	9	—	—
36[a]	18	Ring	176	—	—

[a] N is the number of basis functions, P the number of processors, and $PE = 100\%$ (speedup / P).
[b] These results were obtained when we had sole access to the machine.

For $N = 45$ we tried a number of different values of P, and in every case the measured parallel efficiency (PE = 100% speedup/P) is greater than 90%. This indicates a near-linear speedup with increasing numbers of processors for this problem size. When we looked at larger problem sizes, the PE dropped slightly but still was above 80% in all cases. Due to time constraints, we only ran with 1 and 18 processors problems with 54 and 72 basis functions.

Results with $P = 1$ and varying N reflect performance of the sequential algorithm. This was determined to run at 0.12–0.15 Mflops using our results and the assumption that 1 multiplication = 1.5 addition for scalar processors. [If we use the experimental value of 2.25 for a floating point multiply and add operation (FPMA), then the values vary between 0.11 and 0.14 Mflops.] The experimental Mflops is obtained by simply dividing the total number of additions (after the above conversion) by the total wall clock time taken. While this is a rough estimate, the numbers appear to be reasonable for an Encore Multimax and compares with the 0.14 Mflops that we obtained experimentally from a sequential program that does 50% FPMOs and 50% FPAOs.

As the number of basis functions is increased the wall clock time corresponds reasonably well to estimates [$\mathcal{O}(N^5)$] made based on runs with a similar number of processors but fewer basis functions. This is demonstrated by the results of Table III which show that the ratio of the experimental times to the estimated times is between 1.01 and 1.32. This simply means that the operating system and disk I/O are not significant factors when looking at these problem sizes.

Finally we implemented two guest graphs on runs with 18 processors to see if there would be any effect on the speedup. Our results show that there was not. This is completely consistent with the results of our formulas which predict that 18 processors are not enough to produce a significant difference between communication schemes on the Encore for any reasonable problem size. For smaller problem sizes the precision of our wall clock time measurement was not great enough to see the difference in wall clock time between different communication schemes. For example, the largest difference in communication time between different topologies occurs when comparing a complete graph with a ring when $N = P = 18$. Using the formulas from Table I we estimate that there would only be a difference of about 0.1 s between the two communication times, but we were unable to measure wall clock time to a precision better than 1 s.

Alliant FX-8 Results

Table IV gives the results for $N = 48$ with P being varied on the eight processor Alliant FX-8. As we increase the number of processors from two to eight, the speedups went from 1.6 to 2.9, while the parallel efficiencies varied from 80 to 37%. It is immediately apparent that the speedups and PEs are much less than those attained with roughly the same sized problem on the Encore. When the code is run on one processor, it executes at about 1.9–2.0 Mflops for $N = 48$ and 54. Considering that the code is in general not vectorized, this appears reasonable. (Although we made a small attempt to try to get this to vectorize, we did not pursue this to any great length since it is our purpose to concentrate on the parallelism of algorithms, not on their vectorization.)

We thought that a larger problem size might be needed to obtain better PEs because of the faster processors on the Alliant. Unfortunately, when we ran with $N = 72$ we discovered that due to poor memory access that produced page faults roughly inversely proportional to the number of processors

TABLE III.
Comparison of Timing Estimates and Actual Wall Clock Timings for Encore Multimax.

N[a]	Elapsed Time	Estimated Time	Ratio
36	2434	—	—
45	7472	7428	1.01
54	24430	18483	1.32
72	81881	77888	1.05

[a] N is the number of basis functions.

TABLE IV.
Alliant FX-8 Results for 48 Basis Functions.

P	Time (s)	Speedup	PE (%)[a]
1	660		
2	404	1.6	80
4	268	2.5	62
6	231	2.8	48
8	226	2.9	37

[a] PE = 100% (speedup/P), where P is the number of processors.

used, the code was essentially unusable for problem sizes that employed virtual memory.

Conclusions

We demonstrated by examination of the required computation and interprocessor communication that an implementation of our algorithm on a shared-memory system should give linear speedups for some current machines. At this level of granularity it appears that there is no reason why our algorithm should not run with a linear speedup on a shared-memory machine, if the problem size is such that it is large enough to negate the various types of parallel overhead and if the memory access is improved to prevent undue virtual memory paging.

We have also shown that the implementation of a simple, nonoptimized distributed-memory approach to this problem can produce near linear speedups for at least one shared-memory machine. Complications may arise due to virtual memory paging and the saturation of the shared-memory bus.

It should be emphasized that all of our results have been done with message passing software that was still in the developmental stage. Since the major advantage of our algorithm is that it is portable between both shared- and distributed-memory machines, our future work will concentrate on its implementation on some distributed systems, since it is primarily designed for them. We also want to implement algorithms that utilize more of the permutational symmetry of the integrals and are generalized to include non-load-balanced systems.

In closing it should be noted that our formulas predict that for today's problem sizes the choice of parallel architecture is not very important; but for electronic structure problems of the future some architectures and approaches may be better than others. In particular, high bandwidth, homogenous, distributed-memory architectures and algorithms that can utilize their communication advantages hold great promise. For example, our formulas predict that the load-balanced parallel efficiency of the two most efficient architectures investigated, the complete graph and the parallel communicating hypercube, increases with increasing numbers of processors for the former and is independent of the number of processors used for the latter! While the number of interconnections should make a large scale complete graph impossible, large scale hypercubes can be built.

Acknowledgments

We acknowledge Robert J. Harrison of the Theoretical Chemistry Group of Argonne National Laboratory for many helpful discussions, and the U.S. Department of Energy and the Graduate College of the University of Iowa for partial support of this work.

References

1. As an introduction to the extensive computer science literature on communication algorithms, we recommend: (a) G. S. Almasi and A. Gottlieb, *Highly Parallel Computing*, The Benjamin/Cummings Publishing Company Inc., Redwood City, CA, 1989; (b) *VLSI and Parallel Computation*, R. Suaya and B. Birtwistle, Eds., Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1990.
2. J. Boyle, R. Butler, T. Disz, B. Glickfeld, E. Lusk, R. Overbeek, J. Paterson, and R. Stevens, *Portable Programs for Parallel Processors*, Holt, Rinehart, and Winston, Inc., New York, 1987; J. Paterson, Argonne National Laboratory, private communication 1988.
3. L. A. Covick and K. M. Sando, *J. Comput. Chem.*, **11**, 1151 (1990).
4. R. A. Whiteside, J. S. Binkley, M. E. Colvin, and H. F. Schaefer, III, *J. Chem. Phys.*, **86**, 2185 (1987); M. C. Colvin, Ph.D. thesis, University of California, Berkeley, LBL-23578, DE87 012058, 1986.
5. C. F. Bunge, A. V. Bunge, G. Cisneros, and J. P. Daudey, *Comput. Chem.*, **12**, 91, 109, and 141 (1988). The first of these assumes that all integrals can be held in memory.
6. M. Dupuis and J. D. Watts, *J. Comput. Chem.*, **9**, 158 (1988).
7. J. N. Hurley, D. L. Huestis, and W. A. Goddard, III, *J. Phys. Chem.*, **92**, 4880 (1988).
8. G. Cisneros, C. F. Bunge, and C. C. J. Roothaan, *J. Comput. Chem.*, **8**, 618 (1987).